Fast Abstract: Generating Test cases from Scenario-based Formal Development

Qaisar A. Malik, Johan Lilius and Linas Laibinis
Department of Information Technologies
Åbo Akademi University
Turku Centre for Computer Science (TUCS), Finland
{Qaisar.Malik, Johan.Lilius, Linas.Laibinis}@abo.fi

1. Introduction and Motivation

In this paper, we present an extension of our model-based testing approach based on formal models and user-provided testing scenarios. In this approach, the user provides a testing scenario on the level of an abstract model. When the abstract model is refined to add or modify features, the corresponding testing scenarios are automatically refined to incorporate these changes. Often, due to the abstraction gap between a formal model and the implementation, it is not always feasible to generate implementation code from the formal models. As a result, the implementation is not demonstrated to be correct by its construction but instead it is hand-coded by programmer(s). To validate the correctness of the implementation, testing is performed using user-provided scenarios. The testing scenarios are unfolded into test-cases containing the required inputs and expected outputs. To automate this test-case generation process, we provide guidelines for the formal development of system models. We use Event-B [4, 3] as our formal framework. The main contributions of this work are:

- We provide guidelines for stepwise development of *testable* Event-B models.
- We show how requirements (scenarios) are transformed into the corresponding test-cases.
- We show how inputs and expected outputs for a test-case are derived from Event-B models.

2. Scenario Based Testing

Our model-based testing approach [10, 9] is based on stepwise system development [5] using behavioral models of the system. In such models, the system behavior is modeled as system states together with operations (or events) on the states. In the stepwise development process, an abstract model is first constructed and then further refined to include more details (e.g., functionalities) of the system. Generally, these models can be either formal, informal or both. In this work we only consider formal models.

In the development process, we start with an abstract model M_A and gradually, by a number of refinement steps, obtain a sufficiently detailed concrete model M_C. The final system, the system under test (SUT), is an hand-coded implementation of this detailed model. The left hand-side of the Fig.1 graphically presents this process. Since the

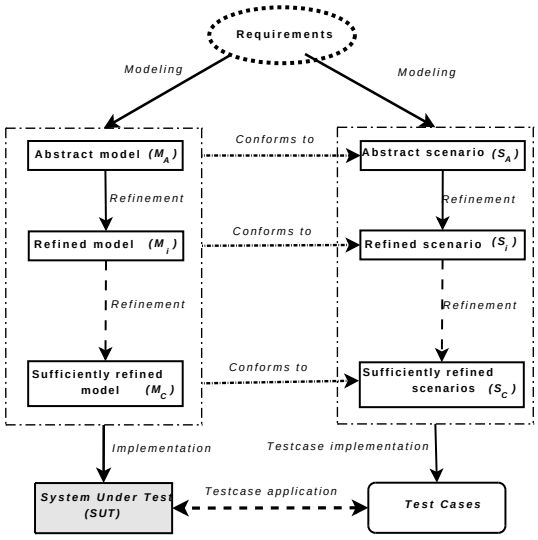

Figure 1: *Overview of our Model-based testing approach*

implementation is no longer *correct-by-construction*, there is need to *test* the implementation. In this paper, we use scenario-based testing [9] to generate the tests. The right hand side of the Fig.1 depicts a parallel process, where we start from the requirements and construct scenarios. A scenario is a description of possible actions and events [1]. In other words, it is one of the expected functionalities of the system. We use the term *test scenario* to emphasize the intended use of these scenarios.

The challenge is now how to refine a test scenario S_A into a concrete test scenario S_C such that S_C covers the same behavior as S_A does. The user-provided scenario S_A is an abstract scenario formally satisfiable ($\models$) by specification model M_A. In the next refinement step, when M_A is refined into M_i, the scenario S_i is constructed automatically from M_A, M_i and S_A in such a way that S_i is formally satisfied by the model M_i. The automatically generated scenario S_i represents functionalities, in part or whole, of the

model M_i.

3 Modeling in Event-B

The Event-B [4, 3] is a recent variation of the classical B-method [2] formalism. Event-B is particularly well-suited for modeling event-based systems. An Event-B specification encapsulates state (variables) of the machine and describes operations (events) on the state. Event-B statements are formally defined using the weakest precondition semantics [6]. We use Event-B as our formal framework.

3.1 Controlled Refinement

In order to automatically refine a scenario from its previous level, we need to use a controlled and structured approach for the refinement of Event-B models. The supported refinement types, for the Event-B, for our testing approach are:

- *Atomicity Refinement*: Where one event operation is replaced by several operations. Intuitively, it corresponds to a branching in the control flow of the system as shown in Fig. 2(a).
- *Superposition Refinement*: Where new implementation details are introduced into the system in the the form of new non-looping or looping events as depicted in Fig. 2(b) and (c) respectively.

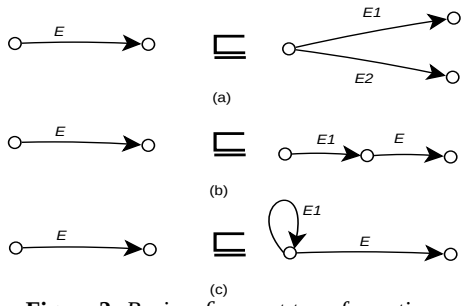

Figure 2: *Basic refinement transformations*

3.2 Classification of Variables and Events

We classify the events and variables of a model into *input*, *output* and *internal* types. The *input events* read inputs from *input variables*. The input events can not write into *output variables*. The *output events* write into the *output* variables. These events are restricted from direct reading of the *input* variables. The *internal events* do not take part in any input/output activity. These may produce intermediate results used by other events.

4 Scenarios and Test cases

We use Communicating Sequential Process (CSP) [7] to represent testing scenarios. The advantage of using CSP is twofold. First, a CSP expression is a convenient way to express several scenarios in a compact form. Second, since we develop our system in a controlled way, i.e. using basic refinement transformations described in Section 3.1, we can associate these Event-B refinements with syntactic transformations of the corresponding CSP expressions. Therefore, knowing the way model M_i was refined by M_{i+1}, we can automatically refine scenario S_i into S_{i+1}. To check whether a scenario S_i is a valid scenario of its model M_i, we use the ProB [8] model checker.

A scenario can be seen as finite sequence of one or more *input-output* units. Where each such unit consists of finite sequence of events with the restriction that each *input-output* unit starts with an *input* event and ends with an *output* event. A *test case t*, when translated from a scenario, consists of a finite sequence of *input-output* units

$$t = \ < U_1, U_2, \ldots, U_n >$$

where each unit specifies the input and expected output of that unit. The motivation for this structure is the following. The developer(s) of the system under test (SUT) may decide to implement the system independent of the structure of a Event-B model. Indeed, it is sometimes hard to follow the strict one to one mapping between events of the model and programming language. For example, two events in a model can be merged to form one operation or the functionality of an event in the model may get divided across multiple operations or classes. However, for successful execution of the system, the interface of the model and its implementation, i.e., the sequence of the inputs and outputs, should remain the same.

5. Conclusions

This work is under progress as doctoral research. In this paper, we presented an abstract view of our scenario-based testing methodology which has successfully been applied on few case study examples. The future directions include: identification of implementation patterns that can aid in automatic testing, the mapping of test cases onto testing frameworks like JUnit, and providing the tool support to automate this testing process.

References

[1] Cambridge Dictionary for English. Available online at http://dictionary.cambridge.org/.

[2] J.-R. Abrial. *The B-Book*. Cambridge University Press, 1996.

[3] J.-R. Abrial. Event Driven Sequential Program Construction. 2000. Available at http://www.matisse.qinetiq.com.

[4] J.-R. Abrial and L.Mussat. Introducing Dynamic Constraints in B. Second International B Conference, LNCS 1393, Springer-Verlag, April 1998.

[5] R.-J. Back and J. von Wright. Refinement calculus, part i: Sequential nondeterministic programs. In *REX Workshop*, pages 42–66, 1989.

[6] E. W. Dijkstra. *A Discipline of Programming*. Prentice-Hall, 1976.

[7] C. A. R. Hoare. *Communicating sequential processes*. Prentice-Hall, Inc., 1985.

[8] M. Leuschel and M. Butler. Prob: A model checker for b. Proc. of FME 2003, Springer-Verlag LNCS 2805, pages 855-874., 2003.

[9] Q. A. Malik, J. Lilius, and L. Laibinis. Model-based testing using scenarios and event-b refinements. Workshop on Methods and Models for Fault-Tolerance(MeMoT), Oxford, UK, 2007.

[10] M. Satpathy, Q. A. Malik, and J. Lilius. Synthesis of scenario based test cases from b models. In *FATES/RV*, pages 133–147, 2006.